15-122: Principles of Imperative Computation, Spring 2016

Homework 8 Programming: Generic Queues

Due: Sunday 3rd April, 2016 by 22:00

For the programming portion of this week's homework, we'll explore a slight variant on the *queue* data structure discussed in class. The challenge of this assignment is primarily adding new functionality to generic queues and then translating this data structure into C.

The code handout for this assignment is at

`qatar.cmu.edu/~srazak/courses/15122-s16/hws/queues-handout.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a **5 handin limit** for this assignment. Additional handins will incur a quarter-point penalty per handin.

1 A Different Implementation of Queues

The function `is_segment(start, end)` used for the queues and stacks in the lecture notes was based on the idea of inclusive/exclusive bounds: the data is stored in the list nodes from `start` (inclusive) to `end` (exclusive).

In this assignment, we'll do things differently and implement queues based on the idea of *inclusive list segments*. Here is how we define them:

- If `start` is `NULL`, then there is an inclusive list segment of length 0 from `start` to `end` (for any value of `end`, in other words, we don't care what `end` is).
- If `start` and `end` are the same and `start->next` is `NULL`, then there is an inclusive list segment of length 1 from `start` to `end`.
- If `start` and `end` are different and there is an inclusive list segment of length $n > 0$ from `start->next` to `end`, then there is an inclusive segment of length $n + 1$ from `start` to `end`.

1.1 Basic Queues (New Data Structure Invariant)

A queue's header node contains three fields, `front`, `back`, and `size`, and it represents a valid queue if there is an inclusive list segment of length `Q->size` from `Q->front` to `Q->back`. This means the queues you will implement for this assignment appear to take one of the two forms in Figure 1, depending on whether or not they are empty.

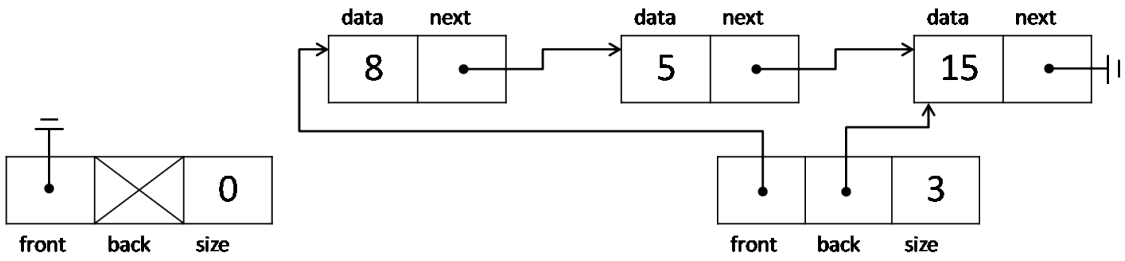

Figure 1: Illustration of the data structure invariants for this assignment.

Figure 1 is inaccurate in one way: our queues will be *generic*, so the `data` field contains data of type **`void`**`*`. Therefore, the data field at the front of the second queue could not actually contain the number 8. At best, it could contain an **`int`**`*`, pointing to allocated memory containing the number 8. Unlike other data structures like hash sets, we *do* allow the generic pointers in a queue to be `NULL`.

Task 1 (2 points) In `queue.c1`, implement the specification function `is_queue(Q)` according to the specification above.

The basic interface for queues is mostly the same as the one in class, extended to be generic by making the elements void pointers. Another difference is that we expose a constant-time function that reports on the size of the queue, since we're storing that information anyway.

Task 2 (2 points) In `queue.c1`, implement the standard queue interface: constant-time functions `queue_new`, `enq`, `deq`, and `queue_size`, for this data structure.

```
// typedef ______* queue_t;

queue_t queue_new()                    /* O(1) */
  /*@ensures \result != NULL; @*/ ;

int queue_size(queue_t Q)              /* O(1) */
  /*@requires Q != NULL; @*/
  /*@ensures \result >= 0; @*/ ;

/* O(1) -- adds an item to the back of the queue */
void enq(queue_t Q, void* x)
  /*@requires Q != NULL; @*/ ;

/* O(1) -- removes an item from the front of the queue */
void* deq(queue_t Q)
  /*@requires Q != NULL && queue_size(Q) > 0; @*/ ;
```

Your functions should work correctly for any data structures that obey the queue data structure invariant.

1.2 Extending the Interface

In this section, we'll implement two additional library functions that, like `queue_size`, could have been implemented by a client using only the four functions above, but that can be implemented in a more efficient way inside the implementation.

```
/* O(i) -- doesn't remove the item from the queue */
void* queue_peek(queue_t Q, int i)
  /*@requires Q != NULL && 0 <= i && i < queue_size(Q); @*/ ;

/* O(n) */
void queue_reverse(queue_t Q)
  /*@requires Q != NULL; @*/ ;
```

The `queue_peek` operation allows the queue to be accessed like an array. Using the second example queue from the introduction, `queue_peek(Q,0)` would return (a pointer to) 8, `queue_peek(Q,1)` would return (a pointer to) 5, and `queue_peek(Q,2)` would return (a pointer to) 15. Peeking at the next-to-be-dequeued element (index 0) should be a constant time operation.

The `queue_reverse` function modifies a queue so that all the elements are in the opposite order they were in before: the old back is the new front, the old front is the new back, and everything in between is switched around. Your implementation of `queue_reverse` should not require you to allocate *any* extra memory. This is only possible because you're extending the library implementation: you'll have to figure out how to reverse all the pointers in the linked list. This is a tricky problem: use loop invariants to guide you!

Task 3 (3 points) In `queue.c1`, implement the linear-time functions `queue_peek(Q,i)` and `queue_reverse(Q)`, for our variant implementation of queues.

Note: These functions *must not allocate memory*, either directly by calling **`alloc`** or indirectly by calling `queue_new` or `enq`.

2 Generic Operations on Generic Queues

The additional C1 features of generic pointers (**void***) and function pointers open up the possibility for new operations on queues that analyze the contents of the queue without modifying (dequeuing from or enqueuing onto) the queue.

2.1 Implementing Generic Interfaces

The first operation we'll ask you to implement is relatively simple. Given a function `prop` that takes an element in the queue and returns `true` or `false`, the function `queue_all(Q,&prop)` checks that `prop` returns `true` on *all* elements of the queue by applying the function to every element one-by-one.

```
typedef bool check_property_fn(void* x);

/* O(n) worst case, assuming P is O(1) */
bool queue_all(queue_t Q, check_property_fn* P)
  /*@requires Q != NULL && P != NULL; @*/ ;
```

Remember that we said $x \geq$ `A[i..j]` was always `true` when $i = j$. The array segment contains no elements, so x is greater than or equal to every one of them! By the same token, if we call `queue_all(Q, &prop)` on an empty queue, we know that the function prop returns `true` on every element in the queue.

The next operation, an *iterator*, is a bit more complicated and a lot more powerful. Iterators take an initial piece of data, the *base case*, as well as a pointer to a function `f`. If the queue `Q` contains the elements `e1`, `e2`, `e3`, and `e4`, then calling `queue_iterate(Q,base,&f)` will compute

```
f(f(f(f(base, e1), e2), e3), e4)
```

whereas if `Q` is empty `queue_iterate(Q,base,&f)` will just return `base`.

```
typedef void* iterate_fn(void* accum, void* x);

/* O(n) worst case, assuming F is O(1) */
void* queue_iterate(queue_t Q, void* base, iterate_fn* F)
  /*@requires Q != NULL && F != NULL; @*/ ;
```

Task 4 (2 points) In `queue.c1`, implement the functions `queue_all` and `queue_iterate` according to the description given above. Neither function should modify the existing queue beyond what their functinal argument may do.

2.2 Using Generic Interfaces

The next task will have you explore generic interfaces by writing short functions that can be passed to generic functions to perform various computations.

Task 5 (4 points) In a new file `queue-use.c1`, implement the following functions, which are intended to be passed to either `queue_all` or `queue_iterate`. Your code in this file should respect the queue interface.

For all of these questions, you can assume that the queue you're working with contains pointers to integers. That is, you can assume `\hastag(int*,x)` will return `true` for every data element in the structure. This means every data element is either `NULL` or an `int*` that has been cast to `void*`. The functions you write should enforce this as a precondition, and should in general have preconditions that ensure safety. Only `incr` should cause any of the data in the queue to be changed.

1. A function `even` such that `queue_all(Q,&even)` returns `true` if all the pointers in `Q` are non-`NULL` and all point to non-negative, even integers.

2. A function `odd` such that `queue_all(Q,&odd)` returns `true` if each pointer in `Q` is either `NULL` or a pointer to positive, odd integers.

3. A function `incr` such that `queue_all(Q,&incr)` always returns `true`, but after the function is run, all the non-`NULL` pointers in the queue should have the integers they point to incremented by 1.

4. A function `find_negative` such that `queue_iterate(Q,NULL,&find_negative)` returns `NULL` if there are no pointers to negative numbers in the queue, and returns the pointer to the negative number closest to the front of the queue if any such element exists. The function should have `\hastag(int*,\result)` as a postcondition.

5. A function `copy` such that `queue_iterate(Q,(void*)queue_new(),&copy)` returns a copy of the queue. The function should have `\hastag(queue_t,\result)` as a postcondition.

6. A function `insert`, which takes two non-`NULL` void pointers that are actually pointers to integers and always returns the first pointer, but that also swaps the integers if the second one is larger than the first one.

The last function you wrote, `insert`, allows you to implement insertion sort! The following code reads out the integers from `Q` one by one and creates a sorted queue `R` with the same integers (though different pointers).

```
queue_t R = queue_new();
while (queue_size(Q) > 0) {
   int* p1 = alloc(int);
   void* p2 = deq(Q);
   //@assert p2 != NULL && \hastag(int*, p2);
   *p1 = *(int*)p2;
   queue_iterate(R, (void*)p1, &insert);
   enq(R,(void*)p1);
}
```

3 Queues in C

For the last part of the assignment, we will turn our C1 queues into a C implementation of queues. Your `queue.c` should begin with at least the following declarations:

```
#include <stdlib.h>         // Standard C library: free(), NULL...
#include <stdbool.h>        // Standard true, false, and bool type
#include "lib/contracts.h"  // Our contracts library
#include "lib/xalloc.h"     // Our allocation library
#include "queue.h"          // The queue interface
```

Here is an *incomplete* list of the changes you will need to make as you adapt your C0/C1 code to C:

- Change calls from **`alloc`** and `alloc_array` to their C analogues. Use the `xalloc` library which defines `xmalloc` and `xcalloc`. These functions abort rather than returning `NULL` when no more memory is available.
- Change `@requires`, `@ensures`, and `@assert` contracts into `REQUIRES()`, `ENSURES()`, and `ASSERT()` C contracts.
- Whenever you are about to lose track of memory (for instance, an allocated list node when dequeuing an element), that memory must be freed.
- Use type `size_t` instead of **`int`** for quantities that are supposed to be array (or queue) offsets; we'll use `size_t` instead of **`int`** for storing the size of the queue in C. Integers of type `size_t` are unsigned, so you don't need to check that they're non-negative.
- Change array types like **`char[]`** to pointers **`char`**$*$. Be careful: this means that you now have to check that arrays are non-NULL! (Although you didn't use arrays in your queue implementation, you may need to be aware of this if you used arrays in any test code you wrote.)

As a stylistic issue, we write

```
int* x = alloc(int);
char[] A = alloc_array(char, 10);
```

in C0/C1 but write

```
int *x = xmalloc(sizeof(int));
char *A = xcalloc(10, sizeof(char));
```

in C. Attaching the $*$ to the variable instead of the type is consistent with the C idea of making the definition of a variable look like the way it is used. We won't be picky about this stylistic issue on this assignment, though.

Task 6 (5 points) Copy the implementation of `queue.c1` to `queue.c`, making sure to include the interface by writing **`#include`** `"queue.h"` within the file `queue.c`. Then adapt your code following the guidelines above so that it is a correct implementation in C.

Handling Deallocation

As is common with C0/C1 to C translations, we have to extend our interface with a function that the client can use to deallocate all the memory reserved for our data structure. (This is a separate issue from any deallocations that may occur as the queue implementation mutates the structure.)

In some cases, the client will want to think of the queue implementation as *owning* the pointers stored in the queue. If the queue owns its data, when we free the queue, we must also cause all the data stored in the queue to be freed. But we can't just call `free` on every pointer in the queue, because we might have a queue of queues or a queue of binary search trees: the client has to specify a function pointer that tells us how to free the data in the queue.

```
typedef void elem_free_fn(void *x);

void queue_free(queue_t Q, elem_free_fn* F)
  /*@requires Q != NULL; @*/ ;
```

If `F` is `NULL`, then we free only the queue's internal data structures, and not the pointers stored in the queue. This means that the queue's data elements are owned elsewhere, and some other data structure has responsibility for freeing the pointers.

Task 7 (2 points) In `queue.c`, implement the `queue_free` function. It should always free **all** the internal memory allocated for the queue itself, and if the function pointer `F` is not `NULL`, then the function it points to should be applied to every `void*` that the client has enqueued in the queue.

None of the functions in your queue interface should leak data: you should check this by writing test cases that free all their data and then running these test programs under `valgrind`.